

EXHIBIT 3

UNITED STATES DISTRICT COURT
NORTHERN DISTRICT OF CALIFORNIA
SAN FRANCISCO DIVISION

ORACLE AMERICA, INC.

Plaintiff,

v.

GOOGLE INC.

Defendant.

Case No. CV 10-03561 WHA

**REPLY EXPERT REPORT OF JOHN C. MITCHELL
REGARDING COPYRIGHT**

**SUBMITTED ON BEHALF OF PLAINTIFF
ORACLE AMERICA, INC.**

TABLE OF CONTENTS

	Page
I. EXPRESSIVE CONTENT IN APIs	1
II. ANDROID'S IMPLEMENTATION OF THE APIs AT ISSUE.....	2
III. EVOLUTION OF THE JAVA APIs.....	7
IV. PARAMETER NAMES	8
V. ORGANIZATION OF THE JAVA APIs	9
VI. C#.....	10
VII. SIGNIFICANCE OF THE COPIED SOURCE CODE FILES	10

1. I, John C. Mitchell, Ph.D., submit the following expert report (“Reply Copyright Report”) on behalf of plaintiff Oracle America, Inc. (“Oracle”). I submit this report in response to the Rebuttal Expert Report of Dr. Owen Astrachan, dated August 12, 2011 (“Astrachan Rebuttal Report”). My Opening and Opposition copyright reports, dated July 29 and August 12, 2011, respectively, already address many of the issues that have been raised by Dr. Astrachan. Rather than repeat the opinions and information contained in my opening report, I incorporate those reports here by reference and will only address my main points of disagreement with the Astrachan Rebuttal Report.

2. In arriving at my opinions provided in this report, I have considered the additional materials referenced in this report, as well as Dr. Astrachan’s Rebuttal Report and attached exhibits, and some of the materials cited therein. I have also considered the reply report of Alan Purdy that analyzes and identifies the private fields in common in the Oracle Java and Android class libraries. I may provide further exhibits to be used as a summary of, or support for, my opinions.

I. EXPRESSIVE CONTENT IN APIs

3. An “API specification” as I have used the term in this case comprises the package names, class and Interface names, method names and signatures (including their arguments, return types, exceptions they may throw), fields, the relationships between these elements, and the prose text that describes them. Dr. Astrachan does not dispute

- that Google engineers had access to the Java API specifications and the Oracle implementation of these specifications;
- the similarity between the Java API specifications (which include the APIs and their descriptions) and the Android API specifications; or
- that the Android implementations of these API specifications contain material taken from Oracle’s specifications.

4. My main disagreement with Dr. Astrachan’s Rebuttal Report concerns whether Oracle’s API specifications contain creative expression. My previous reports explained that they do. I will not repeat the same points that I made previously.

5. I also take issue with Dr. Astrachan’s implication that Android is compatible with the Java APIs. For reasons discussed in, *e.g.*, section IV of my Opposition Report, it is not. For example, one of the earliest Java specification licenses required independent implementations to “implement all the interfaces and functionality of the standard java.* packages as defined by SUN, without subsetting or supersetting.” However, as explained in my Opposition report, Google’s Android is not a complete implementation without subsetting or supersetting.

II. ANDROID’S IMPLEMENTATION OF THE APIs AT ISSUE

6. Dr. Astrachan argues that Google’s implementation of the APIs at issue is not substantially similar to Oracle’s implementation. I have explained otherwise in my previous reports. (*See, e.g.*, my Opening Report section V.I; my Opposition Report section II). The same extensive set of API elements that Android lifts from Oracle’s specifications is also present in Oracle’s implementation.

7. In his Rebuttal Report, Dr. Astrachan states that “we can analyze the differences in the implementations of the APIs by examining the names of the private methods of each implementation.” Astrachan Rebuttal Report ¶ 34. I agree that a comparison of the private method names between implementations can be informative for identifying copying. Private field names and values and private method signatures can be informative on that point as well. What follows Dr. Astrachan’s statement in his Rebuttal Report, however, is not an examination of the names of the private methods of each implementation. Instead, in paragraphs 36-38, Dr. Astrachan describes how he used software to count the number of methods and private methods in each class, determine the percentage of methods that are private in each class, and then average those percentages.

8. Whether or not that is a meaningful calculation, it makes no examination of the *names* of the methods, and so is not the analysis that Dr. Astrachan proposed should be done. Dr. Astrachan's counting calculation sheds very little light on whether Android code was derived from Oracle code. He determines that, among pertinent classes, there are 970 private methods in Android and 1369 private methods in Oracle's JDK 1.5. But if it were the case that all 970 Android private methods were also present in JDK 1.5, with the same names, then that would be strong evidence of derivation. If it were the case that none of the Android private methods were also present in JDK 1.5, then that would point to the opposite conclusion. But Dr. Astrachan's counting analysis cannot tell the difference between the two scenarios. The fact that his numbers for the two implementations are different only means that the two implementations are not identical.

9. Dr. Astrachan says that "My direct inspection of a cross-section of the files at issue confirms the results of this numerical approach." What that cross-section was—which files he looked at, how they were selected, how many there were—he does not say, so it cannot be determined whether the sample set of files he examined was properly representative of the entire Android API implementations. I see that he discusses private methods in connection with the ZipFile class in paragraph 26, but he does not mention the names. There is no good reason to believe that the ZipFile class implementations are representative of the Android and JDK 1.5 API implementations in how private methods are used and named. It is not clear whether the ZipFile class is the only class that Dr. Astrachan inspected directly with respect to private names. In summary, there is no way to tell from Dr. Astrachan's rebuttal report whether the conclusion he drew from examining private method names in particular unspecified files was correct, and there is no way for anyone to independently examine the basis for his ultimate opinion of the lack of substantial similarity.

10. The reply report of Alan Purdy contains an analysis in which he used a program to examine and compare the names of the private fields defined in common classes between Android version 2.2.x (“Froyo,” corresponding to API Level 8) and Oracle’s Java Standard Edition version 5 Development Kit (what Dr. Astrachan refers to as JDK 1.5). Based on the comparison of the private field names, I reach the opposite conclusion as Dr. Astrachan—that there are substantial similarities between the two implementations and the similarities indicate that Android developers had access to and derived from Oracle source code.

11. In 297 classes in common between Android and JDK 1.5 having private fields, Mr. Purdy found 516 private fields with matching names, of which 384 had matching signatures (name, type, and modifier). (By way of illustration, “private static String str” and “private String str” have the same name (str) and type (String), but different modifiers (one is static).) So out of 1,383 total private fields in Android, more than one-third (37.3%) have the same name as in the Oracle code. That suggests derivation.

12. To add to Mr. Purdy’s name comparison, I picked five field names from his list of identical fields and examined the Android and JDK 1.5 source code for the two classes that contain them: javax.crypto.SealedObject and java.security.SignedObject.

13. The class javax.crypto.SealedObject has two private fields having the same type and modifiers in common between Android and JDK 1.5: sealAlg and paramsAlg. When I examined how sealAlg was used in the Android code, I found this:

```

 * Returns the algorithm this object was sealed with.
 *
 * @return the algorithm this object was sealed with.
 * @since Android 1.0
 */
public final String getAlgorithm() {
    return sealAlg;
}

```

14. “sealAlg” and “paramsAlg” are odd choices of names. They are the only names in the SealedObject class that abbreviate “algorithm.” As the purpose of the private field is to store the name of “the algorithm this object was sealed with,” and the method getAlgorithm() is to return that name, one might expect the private field to be named “sealAlgorithm.”

15. The class java.security.SignedObject has four private fields having the same type and modifiers in common between Android and JDK 1.5: serialVersionUID, content, signature, and thealgorithm. “thealgorithm” is an odd name, but in a different way than “sealAlg” and “paramsAlg” from the SealedObject class. I note in passing that the Android and JDK 1.5 versions of SignedObject have a private method named “readObject” in common.

16. The presence of unusual private field names, which do not appear in the public APIs, in both Android and Oracle JDK 1.5 further suggests that Android source code was derived from Oracle source code. This supplements the evidence of access and copying that I discuss in my Opening Report (¶¶ 155-160) and Opposition Report (¶¶ 84-91).

17. There is further evidence of derivation in the private fields. Classes that implement the Serializable Interface may include a private field called “serialVersionUID” that helps identify the version of the class of a serialized object. It can be any arbitrary long integer, but according to Oracle’s Java Object Serialization Specification (version 1.5.0), it should be a 64-bit hash of the class name, interface class names, methods, and fields. *See*

<http://download.oracle.com/javase/1.5.0/docs/guide/serialization/spec/class.html#4100>.

Accordingly, if there were differences between the Android and Oracle Java implementations of a class, one would expect that they would have different values for their serialVersionUID fields. But this is not the case. This table shows the values for the serialVersionUID fields in a few classes implemented in Oracle JDK 1.5 and Android:

JDK 1.5 (J2SE 5.0) serialVersionUID	Android 2.2 (Froyo) serialVersionUID
java.io.File.java:	java.io.File.java:
301077366599181567L	301077366599181567L
java.security.SignedObject.java:	java.security.SignedObject.java:
720502720485447167L	720502720485447167L
java.util.Date.java:	java.util.Date.java:
7523967970034938905L	7523967970034938905L
java.lang.String.java:	java.lang.String.java:
-6849794470754667710L	-6849794470754667710L
java.net.Inet4Address.java:	java.net.Inet4Address.java:
3286316764910316507L	3286316764910316507L

18. As is readily apparent, the private fields are set to the same values in Android and Oracle Java. From this one can conclude that either the class name, interface class names, methods, and fields are exactly the same between the two implementations (so the hash would be the same) or the Android developers took the values from the Oracle source code. Either case is consistent with Android being derived from Oracle Java.

III. EVOLUTION OF THE JAVA APIs

19. Dr. Astrachan states that “When new versions are released, API elements are essentially never changed or removed, only added.” However, this characterization overlooks at least one additional option that is available to API designers: an API element may be deprecated. As a result of deprecation, the relation between one version of the Java API and subsequent versions of the Java API may be more complex.

20. The Java SE Compatibility Policy that Dr. Astrachan cites (available at <http://java.sun.com/j2se/1.5.0/compatibility.html>.) explains that “Deprecated APIs are interfaces that are supported only for backwards compatibility.” *Id.* In addition, “It is recommended that programs be modified to eliminate the use of deprecated APIs, though these APIs may remain in the API specification and its implementation.” *Id.* In other words, deprecated elements may be used in code, but the API designers recommend against it. As an aid to programmers who wish to use only recommended portions of the API, the “javac compiler generates a warning message whenever one of these is used, unless the -nowarn command-line option is used.” *Id.*

21. There are over one hundred, and possibly two to three hundred, deprecated portions of past Java APIs listed and explained at, *e.g.*, <http://download.oracle.com/javase/1.5.0/docs/api/deprecated-list.html>. The documentation for the class `java.security.Signer`, for example, indicates that it is “no longer used” and that “[i]ts functionality has been replaced” by other classes. (<http://download.oracle.com/javase/1.5.0/docs/api/java/security/Signer.html>) Significantly, despite Oracle’s statement that this class’s “functionality has been replaced,” Android still includes it as part of its API with the comment “[t]his class is deprecated.” (<http://developer.android.com/reference/java/security/Signer.html>).

22. Because deprecating elements of an API does not prevent them from being used, there is no functional compatibility reason for Google to retain this Oracle designation in its copy of relevant portions of the Java API. Nonetheless, Google appears

to have copied the portions of the Java API specification that say which elements are deprecated.

23. The inclusion in Android of classes whose functionality has been replaced indicates that Google took more than what was necessary to achieve functional compatibility with Java. For additional examples of classes and other elements in Android that are deprecated in the Java API, see, e.g.,

<http://developer.android.com/reference/java/security/Identity.html>;

<http://developer.android.com/reference/java/security/IdentityScope.html>;

<http://developer.android.com/reference/java/io/LineNumberInputStream.html>;

http://developer.android.com/reference/java/net/HttpURLConnection.html#HTTP_SERV_ER_ERROR. See also the discussion of the evolution of java.util in my Opposition Report at ¶¶ 33-35.

IV. PARAMETER NAMES

24. Dr. Astrachan’s Rebuttal Report does not address my discussion of the selection and order of parameters. (See, e.g., my Opening Report ¶¶ 185-87; my Opposition Report ¶¶ 22, 44). Instead, he questions the creativity of parameter names. Astrachan concedes that parameter names “need not be reused by programmers.” (Astrachan Rebuttal Report ¶ 42). Software using an API may call methods defined in the API without any dependence on the way parameters are named in the API specification.

25. Dr. Astrachan nevertheless calls parameter names functional, in that they “serve to inform programmers what kind of information the method expects.” *Id.* Dr. Astrachan mistakes the capacity to be descriptive with functionality. Under his definition of “functional,” almost any descriptive textual element is functional, because it would provide the “function” of describing something. My previous reports already explain that parameter names are not functional and can be and often are quite original. (See, e.g., my Opposition Report ¶ 43-44).

V. ORGANIZATION OF THE JAVA APIs

26. The hierarchical organization of packages, classes, Interfaces, and other organizational aspects of APIs are extensively analyzed and addressed in my Opposition Report so I will not repeat that information here. (*See, e.g.*, my Opposition Report section I.B.).

27. Contrary to Dr. Astrachan's argument, the choice to follow a design pattern (*see* Astrachan Rebuttal Report ¶ 45) is hardly a design "rule," and even within a particular pattern there are many choices. Multiple authorities support this point, which I discuss below.

28. One explanation of design patterns and their relation to the design of software libraries and the systems that use them appears in the Foreword of the seminal "gang of four" book that Dr. Astrachan cites:

All well-structured object-oriented architectures are full of patterns. Indeed, one of the ways I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects. Focusing on such mechanisms during systems development can yield an architecture that is smaller, simpler, and far more understandable than if these patterns are ignored. – Grady Booch.

Erich Gamma et al., DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE xiii (1994). While this quote explains the value of design patterns for achieving smaller, simpler, and more understandable software, it in no way suggests that their use is mandatory, or that design patterns routinely or forcibly produce good designs.

29. The importance of skill, creativity, and careful thought is also emphasized in the introduction to the book, which begins as follows:

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get ‘right’ the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Id. at 1. Note that the purpose of a Java library is to provide reusable object-oriented software.

30. The book quotes Christopher Alexander, who said, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” *Id.* at 2. While this statement was originally written about architecture (of buildings), the authors of *Design Patterns* state that the same description also applies to software. I agree. The quote illustrates the difference between idea and expression: the same technical solution can be expressed in many different, original ways. Any one particular way is the creative expression of the API designer who designed it.

VI. C#

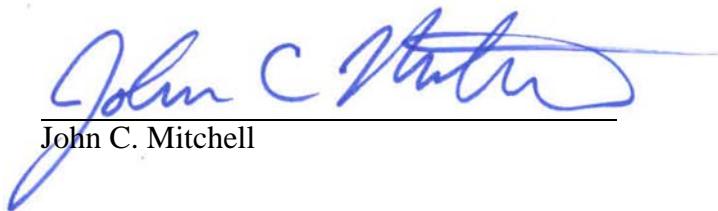
31. In his discussion of the C# language, the only new point Dr. Astrachan raises is meaning of “proprietary.” His argument does not affect the overall conclusions in ¶ 121 of my Opening Report.

VII. SIGNIFICANCE OF THE COPIED SOURCE CODE FILES

32. Section III of my Opposition Report discussed the significance of the copied source code files, and I will not repeat those arguments here. Again, Dr. Astrachan’s report does not dispute that Google copied eight entire Oracle class files,

each of which can be considered an original program in itself, and code and comments from three other Oracle files.

Dated August 19, 2011



John C. Mitchell